Simple Optimizations

March 30, 2011
CS 132: Compiler Design
# Instruction Scheduling

Form of latency hiding.

1. \( t_2 \leftarrow *t_1 \)
2. \( t_3 \leftarrow t_2 + 4 \)
3. \( t_1 \leftarrow *t_4 \)
4. \( t_5 \leftarrow *(t_4 + 4) \)
5. \( t_5 \leftarrow t_1 + t_5 \)
6. \( t_6 \leftarrow *(t_4 + 8) \)
7. \( t_5 \leftarrow t_2 + t_6 \)
8. \( t_4 \leftarrow t_5 + 12 \)
9. \( *t_8 \leftarrow t_7 \)
Dependencies

Two instructions $s_1$ and $s_2$ are have a dependency if there is a constraint on their relative order.

Dependencies in a basic block can be represented as a DAG:

- An edge from $s_1$ to $s_2$ if there is a dependency between them.
- Often, edges are labeled with latency information.
Dependencies

\( s_2 \) depends on \( s_1 \) if:

- \( s_1 \) writes a value that …
- \( s_1 \) writes a value that …
- \( s_1 \) reads a value that …
- \( s_2 \) is a jump that …
**Dependencies**

$s_2$ depends on $s_1$ if:

- ✔️ If $s_1$ writes a value that …
- ✔️ If $s_1$ writes a value that …
- ✔️ If $s_1$ reads a value that …
- ✔️ If $s_2$ is a jump that …

What is the dependency DAG for the previous page? (Assume loads take two cycles.)
Any topological sort of the dependence graph yields a valid ordering.
Scheduling

Any topological sort of the dependence graph yields a valid ordering

So, we just want to find the topological sort which minimizes execution time.
SCHEDULING

Any topological sort of the dependence graph yields a valid ordering.

So, we just want to find the topological sort which minimizes execution time.

Guess how easy this is?
Scheduling

Any topological sort of the dependence graph yields a valid ordering.

So, we just want to find the topological sort which minimizes execution time.

Guess how easy this is?
**Greedy Heuristic: (Forward) List Scheduling**

Assigns priorities based on max distance from a leaf.
Greedy Heuristic: (Forward) List Scheduling

Assigns priorities based on max distance from a leaf.

Repeatedly:

✓ Choose a ready instruction with the highest priority.
✓ Or, schedule an instruction whose predecessors have all been chosen and live with the stall.
Greedy Heuristic: (Forward) List Scheduling

Assigns priorities based on max distance from a leaf.

Repeatedly:

✓ Choose a ready instruction with the highest priority.

✓ Or, schedule an instruction whose predecessors have all been chosen and live with the stall.

Exercise: List-schedule the code.
In general, code cannot be re-ordered if it would affect effects.

For example, in SML, which of the following pairs of lines can be swapped?

\[
\begin{align*}
\text{val } x &= a \ast b \\
\text{val } y &= c \ast d \\
\text{val } x &= a \div b \\
\text{val } y &= c \div d
\end{align*}
\]
Peehole Optimization

Run through the code looking for specific instruction sequences to optimize:

```
movl %ebx, %ebx
addl $0, %eax
addl %ebx, %edx
addl $4, %edx
addl $8, %edx
jmp L1
L1:
   imull $4, %edx
   imull $6, %eax
```
**Peephole Optimization**

Run through the code looking for specific instruction sequences to optimize:

```assembly
movl %ebx, %ebx
addl $0, %eax
addl %ebx, %edx
addl $4, %edx
addl $8, %edx
jmp L1
L1:
imull $4, %edx
imull $6, %eax
```

(What about division by 2?)
32-bit Signed Division of $r$ by 2
32-bit Signed Division of \( r \) by 2

1. Add 1 if \( r \) is negative.
**32-bit Signed Division of r by 2**

1. Add 1 if \( r \) is negative.
   - E.g., \( r \leftarrow r + (r \uparrow \gg 31) \)

2. Arithmetic shift right \((r \downarrow \gg 1)\).
32-bit Signed Division of $r$ by 2

1. Add 1 if $r$ is negative.
   - E.g., $r \leftarrow r + (r \gg 31)$

2. Arithmetic shift right ($r \gg 1$).

In general, to divide by $2^k$,
1. Add $2^k - 1$ to $r$ if $r$ is negative
2. Arithmetic shift right ($r \gg k$).
32-bit Signed Division of \( r \) by 3
32-bit Signed Division of $r$ by 3

- Get high 32 bits of 64-bit product
  
  $$r \times 0x55555556$$

- Increment result if $r$ was negative
**CONSTANT FOLDING / CONSTANT PROPAGATION**

```c
int i = 3 + 8;
double d1 = sqrt(4.0);
double d2 = 1.0 / 3.0;
double d3 = 3.0 / 1.0;
double f = d3 / (d3 - 3.0);
int n = i / (i-11);

i = 12;
y = *(p+i);
x = i + 9;
```
Algebraic Simplification

```plaintext
int i1 = ...;
int i2 = i1 * 0 + i1;
int i3 = (4 + i2) - 2;
bool b1 = (i1 != i1);

double d1 = ...;
double d2 = (x + d1) - d1;
double d3 = d1 * 0.0;
double d4 = d1 + 0.0;
bool b2 = (d1 != d1);
```
**Algebraic Simplification**

```c
double oldeps;
double eps = 1.0;
while (eps + 1.0 > 1.0) {
    oldeps = eps;
    eps = 0.5 * eps;
}
```
**Algebraic Simplification**

double oldeps;
double eps = 1.0;
while (eps + 1.0 > 1.0) {
    oldeps = eps;
    eps = 0.5 * eps;
}

double oldeps;
double eps = 1.0;
while (eps > 0.0) {
    oldeps = eps;
    eps = 0.5 * eps;
}
Copy Propagation

\[
x = y; \\
z = 2\times x; \\
w = 2\times y;
\]
**Dead Code Elimination**

\[
\begin{align*}
w &= y; \\
z &= 2*y; \\
w &= 2*y;
\end{align*}
\]

\[
\begin{align*}
\text{while (1); \\
\text{launchNuclearMissile(); \\
\text{return;}
\end{align*}
\]
Loop Unrolling

What is the loop unrolling optimization? Why is it useful?
Is Loop Unrolling Really Helpful?

L1: x = *i
    s += x
    i += 4
    if (i<n) goto L1

↓

L1: x = *i
    s += x
    i += 4
    if (i>=n) goto L3

L2: x = *i
    s += x
    i += 4
    if (i<n) goto L1

L3:
Better?

L1: x = *i
    s += x
    i += 4
    if (i<n) goto L1

↓

L1: x = *i
    s += x
    x = *(i+4)
    s += x
    i += 8
    if (i<n) goto L1
Better?

L1: \( x = *i \)
    \( s += x \)
    \( i += 4 \)
    if \( (i<n) \) goto L1

↓

if \( i > n-8 \) goto L2

L1: \( x = *i \)
    \( s += x \)
    \( x = *(i+4) \)
    \( s += x \)
    \( i += 8 \)
    if \( (i<n-8) \) goto L1

L2: \( x = *i \)
    \( s += x \)
    \( i += 4 \)
    if \( (i<n) \) goto L2